

Analyzing Expected Outcomes and Almost-Sure Termination of Probabilistic Programs is Hard*

Benjamin Lucien Kaminski

Joost-Pieter Katoen

October 28, 2014

Abstract

This paper considers the computational hardness of computing expected outcomes and deciding almost-sure termination of probabilistic programs. We show that deciding almost-sure termination and deciding whether the expected outcome of a program equals a given rational value is Π_2^0 -complete. Computing lower and upper bounds on the expected outcome is shown to be recursively enumerable and Σ_2^0 -complete, respectively.

1 Introduction

Probabilistic programs [10] are imperative sequential programs with the ability to draw values at random from probability distributions. They are used in security to describe cryptographic constructions (such as randomized encryption) and security experiments [2], in machine learning to describe distribution functions that are analyzed using Bayesian inference [3], and in randomized algorithms. They are typically just a small number of lines, but hard to understand and analyze, let alone algorithmically.

This paper considers a precise classification of the computational hardness of solving two main analysis problems for probabilistic programs: (1) almost-sure termination [8] — does a program terminate with probability one? — and (2) computing expected outcomes — is the expected outcome of a program (variable) equal, smaller, or larger than a given rational number? Expected outcomes correspond to McIver & Morgan’s weakest pre-expectation semantics of pGCL, the probabilistic version of Dijkstra’s guarded command language [11].

Several results on computational hardness in connection with analyzing probabilistic programs have been reported in the literature, like non-recursive-enumerability results for probabilistic rewriting logic [4] and decidability results for restricted probabilistic programming languages [13]. A lot of work has also been done towards automated reasoning for almost-sure termination. For instance [17] gives an overview of some particularly interesting examples of probabilistic logical programs and the according intuition for proving almost-sure termination. Arons *et al.* reduce almost-sure termination to termination of a non-deterministic program by means of a planner [1]. In [6], a pattern-based approach which exploits this idea together with a prototypical tool support is presented.

*This research is funded by the Excellence Initiative of the German federal and state governments and by the EU FP7 MEALS project.

Despite several approaches to tackle the problem of almost-sure termination in an automated manner, the majority of the literature does not consider the hardness of the problem or states that it must intuitively be harder to solve than the termination problem for ordinary, i.e., non-probabilistic programs. For instance in [12] it is noted that while partial correctness for small-scale examples is not harder to prove than for ordinary programs, the case for total correctness of a probabilistic loop must be harder to analyze. As another example [6] suggests that almost-sure termination must be harder to decide than ordinary termination since for the latter a topological argument suffices while for the former arithmetical reasoning is needed.

Aside from the intuition that almost-sure termination must be somewhat harder to decide, to the best of our knowledge, there seems to be yet no *precise* classification of the computational hardness of deciding this major analysis problem. This gap is bridged by this paper. Such a precise classification is not only of theoretical interest, but allows for deeper insights into the specific difficulties of dealing with the problem of almost-sure termination and may help in identifying subclasses of programs for which it becomes easier to solve. Through our classification we cannot only establish that almost-sure termination is in fact strictly harder to decide than ordinary termination but we can also make a statement on the upper bound of the hardness of the problem.

In this paper we study and formalize the problem of computing expected outcomes and the problem of deciding almost-sure termination, and we establish the following hardness results: We first show that computing lower bounds on the expected outcome of program variable v by executing a probabilistic program P is recursively enumerable. Computing upper bounds for the expected outcome is shown to be Σ_2^0 -complete, whereas deciding whether the expected outcome of v equals some rational is shown to be Π_2^0 -complete. Finally, almost-sure termination of P is shown to be Π_2^0 -complete. The immediate consequences of the latter result are twofold: (1) deciding almost-sure termination is not only intuitively but provably strictly harder than deciding termination for ordinary programs, and (2) deciding almost-sure termination is *not* harder than deciding whether an *ordinary* program halts on all or infinitely many inputs [15].

Regarding the consequences of the Σ_2^0 -completeness of computing upper bounds for expected outcomes we note the following: If both upper and lower bounds for expected outcomes were recursively enumerable, then each expected outcome would be a computable real number. However, since upper bounds are shown to be not recursively enumerable (implied by the Σ_2^0 -completeness), we can establish that in general it is not possible to approximate expected outcomes from above and from below with arbitrary precision.

Our hardness results are established by reductions from the universal halting problem for ordinary programs. Remarkably the probabilistic programs we use in our reduction obey a certain syntactic schema, namely that the randomization and the actual computation are strictly separated. We interpret this as possible evidence for the existence of a “normal form” for probabilistic programs.

2 Preliminaries

In order to have a frame of reference in which we can classify the hardness of calculating expected outcomes of probabilistic programs, we first briefly recall the concept of the arithmetical hierarchy:

Definition 1 (ARITHMETICAL HIERARCHY [9, 14]):

The **class** Σ_n^0 is defined as

$$\Sigma_n^0 = \left\{ \mathcal{A} \mid \mathcal{A} = \{ \vec{x} \mid \exists y_1 \forall y_2 \exists y_3 \cdots \exists / \forall y_n : (\vec{x}, y_1, y_2, y_3, \dots, y_n) \in \mathcal{R} \}, \right. \\ \left. \mathcal{R} \text{ is a decidable relation} \right\},$$

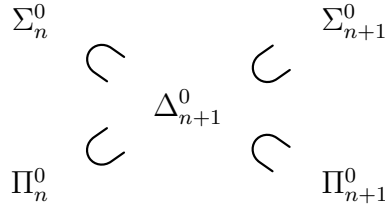
the **class** Π_n^0 is defined as

$$\Pi_n^0 = \left\{ \mathcal{A} \mid \mathcal{A} = \{ \vec{x} \mid \forall y_1 \exists y_2 \forall y_3 \cdots \exists / \forall y_n : (\vec{x}, y_1, y_2, y_3, \dots, y_n) \in \mathcal{R} \}, \right. \\ \left. \mathcal{R} \text{ is a decidable relation} \right\},$$

and the **class** Δ_n^0 is defined as $\Delta_n^0 = \Sigma_n^0 \cap \Pi_n^0$, for every $n \in \mathbb{N}$.

Note that we implicitly always quantify over \mathbb{Q}^+ and that by the \vec{x} 's we mean tuples over \mathbb{Q}^+ . *Multiple consecutive quantifiers of the same type* can be contracted to *one* quantifier of that type, so the number n really refers to the number of necessary *quantifier alternations* rather than to the number of quantifiers used.

A set \mathcal{A} is called **arithmetical**, iff $\mathcal{A} \in \Gamma_n^0$, for some $\Gamma \in \{\Sigma, \Pi, \Delta\}$ and some $n \in \mathbb{N}$. The inclusion diagram



holds for every $n \geq 1$, thus the arithmetical sets form a strict hierarchy. Furthermore note that $\Sigma_0^0 = \Pi_0^0 = \Delta_0^0 = \Delta_1^0$ is exactly the class of the decidable sets and Σ_1^0 is exactly the class of the recursively enumerable sets.

Next we recall the concept of many-one reducibility and the concept of completeness. Both these notions allow us to precisely classify the hardness of calculating expected outcomes and deciding almost-sure termination.

Definition 2 (MANY-ONE REDUCIBILITY [14, 16]):

Let \mathcal{A}, \mathcal{B} be arithmetical and let X be some appropriate universe, such that $\mathcal{A}, \mathcal{B} \subseteq X$. A set \mathcal{A} is called **many-one-reducible** to a set \mathcal{B} , denoted $\mathcal{A} \leq_m \mathcal{B}$, iff there exists a computable function $f: X \rightarrow X$, such that

$$\forall \vec{x} \in X: (\vec{x} \in \mathcal{A} \iff f(\vec{x}) \in \mathcal{B}).$$

If f is a function, such that f many-one reduces \mathcal{A} to \mathcal{B} , we denote this by $f: \mathcal{A} \leq_m \mathcal{B}$. Note that \leq_m is obviously transitive.

Definition 3 (Γ_n^0 -COMPLETENESS [14]):

A set \mathcal{A} is called **Γ_n^0 -complete**, for $\Gamma \in \{\Sigma, \Pi, \Delta\}$, iff both $\mathcal{A} \in \Gamma_n^0$ and \mathcal{A} is **Γ_n^0 -hard**, meaning $\mathcal{B} \leq_m \mathcal{A}$, for any set $\mathcal{B} \in \Gamma_n^0$.

An important fact about Σ_n^0 - and Π_n^0 -complete sets is that they are in some sense the most complicated sets in Σ_n^0 and Π_n^0 , respectively. Formally, this can be expressed as follows:

Lemma 1 (PROPERTIES OF COMPLETE SETS [5]):

If \mathcal{A} is Σ_n^0 -complete, then $\mathcal{A} \in \Sigma_n^0 \setminus \Pi_n^0$. Analogously if \mathcal{A} is Π_n^0 -complete, then $\mathcal{A} \in \Pi_n^0 \setminus \Sigma_n^0$.

Lemma 1 implies in particular that for a Σ_n^0 -complete set \mathcal{A} it holds that $\mathcal{A} \notin \Delta_n^0$.

3 Probabilistic Programs

In order to speak about probabilistic programs and the computations performed by such programs, we first briefly introduce their syntax and their semantics:

Definition 4 (SYNTAX OF PROBABILISTIC PROGRAMS):

Let **Var** be the **set of program variables**. The **set Prog of probabilistic programs** is defined inductively as follows: For any $v \in \mathbf{Var}$ and any arithmetical expression e over **Var** (not to be confused with the sets from the arithmetical hierarchy), the assignment $v := e$ is in **Prog**. Furthermore if $P_1, P_2 \in \mathbf{Prog}$, $p \in [0, 1] \subseteq \mathbb{Q}$, and if b is a Boolean expression over arithmetic expressions then the concatenation $P_1; P_2$, the probabilistic choice $\{P_1\} [p] \{P_2\}$, and the while-loop **WHILE** (b) $\{P_1\}$ are also in **Prog**. We call the set of programs that *do not* contain any probabilistic choices the **set of ordinary programs** and denote this set by **ordProg**.

This syntax is a subset of pGCL originated from McIver and Morgan [11]. We omitted **skip**-, **abort**-, and **if**-statements, as those are syntactic sugar. Furthermore, we do not consider (non-probabilistic) non-determinism. The operational semantics for our programs is given below:

Definition 5 (SEMANTICS OF PROBABILISTIC PROGRAMS):

Let the set of variable valuations be denoted by $\mathbb{V} = \{\eta \mid \eta: \mathbf{Var} \rightarrow \mathbb{Q}^+\}$, let the set of program states be denoted by $\mathbb{S} = (\mathbf{Prog} \cup \{\downarrow\}) \times \mathbb{V} \times I \times \{L, R\}^*$, for $I = [0, 1] \subseteq \mathbb{Q}^+$, let $\llbracket e \rrbracket_\eta$ be the evaluation of the arithmetic expression e given the variable valuation η , and analogously let $\llbracket b \rrbracket_\eta$ be the evaluation of the Boolean expression b . Then the **semantics of probabilistic programs** is given by the smallest relation $\vdash \subseteq \mathbb{S} \times \mathbb{S}$ which satisfies the following inference rules:

$$\begin{aligned}
& (\text{assign}) \frac{}{\langle v := e, \eta, a, \theta \rangle \vdash \langle \downarrow, \eta[v \mapsto \max\{\llbracket e \rrbracket_\eta, 0\}], a, \theta \rangle} \\
& (\text{concat1}) \frac{\langle P_1, \eta, a, \theta \rangle \vdash \langle P'_1, \eta', a', \theta' \rangle}{\langle P_1; P_2, \eta, a, \theta \rangle \vdash \langle P'_1; P_2, \eta', a', \theta' \rangle} \\
& (\text{concat2}) \frac{}{\langle \downarrow; P_2, \eta, a, \theta \rangle \vdash \langle P_2, \eta, a, \theta \rangle} \\
& (\text{prob1}) \frac{}{\langle \{P_1\} [p] \{P_2\}, \eta, a, \theta \rangle \vdash \langle P_1, \eta, a \cdot p, \theta \cdot L \rangle} \\
& (\text{prob2}) \frac{}{\langle \{P_1\} [p] \{P_2\}, \eta, a, \theta \rangle \vdash \langle P_2, \eta, a \cdot (1 - p), \theta \cdot R \rangle} \\
& (\text{while1}) \frac{\llbracket b \rrbracket_\eta = \text{True}}{\langle \text{WHILE } (b) \{P\}, \eta, a, \theta \rangle \vdash \langle P; \text{WHILE } (b) \{P\}, \eta, a, \theta \rangle}
\end{aligned}$$

$$(\text{while2}) \frac{\llbracket b \rrbracket_\eta = \text{False}}{\langle \text{WHILE } (b) \{P\}, \eta, a, \theta \rangle \vdash \langle \downarrow, \eta, a, \theta \rangle}$$

We use $\sigma \vdash^k \tau$ and $\sigma \vdash^* \tau$ in the usual sense. Furthermore we write $\sigma \vdash_{(\text{name})} \tau$ if τ is inferred by the use of the (name)–rule (for $\text{name} \in \{\text{assign}, \text{concat1}, \dots\}$).

The semantics is mostly straightforward except for two things: in addition to the program that is to be executed next and the current variable valuation, each state also stores a string $\theta \in \{L, R\}^*$ that indicates which probabilistic choices were made in the past (**L**eft or **R**ight) as well as the probability a with which those choices were made. The graph that is spanned by the \vdash –relation can be seen as an unfolding of the MDP–semantics for pGCL provided by Gretz *et al.* [7].

4 Expected Outcomes and Termination Probabilities

In this section we formally define the notions of the expected outcome of $P \in \text{Prog}$ as well as its termination probability. We start by some auxiliary notions: It is a well-known result due to Kleene that for any program state σ for which the successor is inferred without the use of the (prob1)– or the (prob2)–rule, i.e., the next instruction to be executed is *not* a probabilistic choice, the successor of σ is unique and computable:

Theorem 1 (THE STATE SUCCESSOR FUNCTION T [9]):

Let \mathbb{S}_o be the set of program states for which the successor is inferred without the use of the (prob1)– or the (prob2)–rule. Then there exists a total computable function $T: \mathbb{S}_o \rightarrow (\mathbb{S} \cup \{\top\})$, such that

$$T(\sigma) = \begin{cases} \tau, & \text{if } \sigma \vdash \tau \\ \top, & \text{if } \sigma = \langle \downarrow, \eta, a, \theta \rangle. \end{cases}$$

The successor of a state $\sigma \in \mathbb{S} \setminus \mathbb{S}_o$ is not unique, because the program chooses a left or a right branch with some probability. However, if we resolve the probabilistic choice by providing a symbol L or R that indicates whether the left or the right branch shall be chosen, we can come up with a computable function T_{prob} which computes a unique successor:

Corollary 1 (THE PROBABILISTIC STATE SUCCESSOR FUNCTION T_{prob}):

There exists a total computable function $T_{\text{prob}}: (\mathbb{S} \setminus \mathbb{S}_o) \times \{L, R\} \rightarrow \mathbb{S}$, such that

$$T_{\text{prob}}(\sigma, s) = \begin{cases} \tau_L, & \text{if } s = L \text{ and } \sigma \vdash_{(\text{prob1})} \tau_L \\ \tau_R, & \text{if } s = R \text{ and } \sigma \vdash_{(\text{prob2})} \tau_R. \end{cases}$$

While T and T_{prob} each compute only the next successor state, we can also define a computable function T_{prob}^* that computes the k -th successor state according to some sequence $w \in \{L, R\}^*$ which tells T_{prob}^* how to resolve the probabilistic choices that occur along the computation:

Corollary 2 (THE k -TH STATE SUCCESSOR FUNCTION T_{prob}^*):

There exists a total computable function $T_{\text{prob}}^: \mathbb{S} \times \mathbb{N} \times \{L, R\}^* \rightarrow (\mathbb{S} \cup \{\top\})$,*

such that

$$T_{prob}^*(\sigma, k, w) = \begin{cases} \tau, & \text{if } \sigma = \langle P, \eta, a, \theta \rangle \vdash^k \langle P', \eta', a', \theta \cdot w \rangle = \tau \\ \top, & \text{otherwise.} \end{cases}$$

So T_{prob}^* returns a successor state τ , if $\sigma \vdash^k \tau$, whereupon exactly $|w|$ inferences must use the (prob1)– or the (prob2)–rule and the probabilistic choices are resolved according to w . Otherwise T_{prob}^* returns \top . Note in particular that for both the inference of a terminal state $\langle \downarrow, \eta, a, \theta \rangle$ within less than k steps as well as the inference of a k -th successor state through less than $|w|$ probabilistic choices, the calculation of T_{prob}^* will result in \top .

In addition to T_{prob}^* , we will need three more computable operations for defining the expected outcomes and almost-sure termination:

Corollary 3:

There exist total computable functions $\alpha: (\mathbb{S} \cup \{\top\}) \rightarrow \mathbb{Q}^+$, $\wp: (\mathbb{S} \cup \{\top\}) \times \mathbf{Var} \rightarrow \mathbb{Q}^+$, and $h: \mathbb{N} \rightarrow \{L, R\}^*$, such that

$$\alpha(\sigma) = \begin{cases} a, & \text{if } \sigma = \langle \downarrow, \eta, a, \theta \rangle \\ 0, & \text{otherwise,} \end{cases} \quad (1)$$

$$\wp(\sigma, v) = \begin{cases} \eta(v) \cdot a, & \text{if } \sigma = \langle \downarrow, \eta, a, \theta \rangle \\ 0, & \text{otherwise,} \end{cases} \quad (2)$$

$$h \text{ is a computable bijection.} \quad (3)$$

So the function α takes a state σ and returns the probability of reaching σ , and the function \wp takes a state σ and a variable v and returns the probability of reaching σ multiplied with the value of v in the state σ . Both functions do that only if the provided state σ is a terminal state. Otherwise they return 0.

We now have all the concepts and notations available for defining the expected outcome of a program variable after executing a probabilistic program and for defining the program's termination probability:

Definition 6 (EXPECTED OUTCOMES AND TERMINATION PROBABILITIES):

Let $P \in \mathbf{Prog}$ and let $\eta_0 \in \mathbb{V}$, such that $\forall x \in \mathbf{Var}: \eta_0(x) = 0$. Then starting in η_0 ,

1. the **expected outcome** of $v \in \mathbf{Var}$ after executing P , denoted $\mathbf{E}_P(v)$, is given by

$$\mathbf{E}_P(v) := \sum_{i \in \mathbb{N}} \sum_{j \in \mathbb{N}} \wp(T_{prob}^*(\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v),$$

2. the **probability that P terminates**, denoted $\mathbf{Pr}_P(\downarrow)$, is given by

$$\mathbf{Pr}_P(\downarrow) := \sum_{i \in \mathbb{N}} \sum_{j \in \mathbb{N}} \alpha(T_{prob}^*(\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i))).$$

$\mathbf{E}_P(v)$ is basically equivalent to the expected reward introduced by Gretz *et al.* in [7] and thereby coincides with the expectation transformer semantics by McIver and Morgan

[11]. The main difference is that the expected reward is defined on a Markov decision process, whereas $E_P(v)$ is defined on its unfolding. In principle, for $E_P(v)$ we sum over all possible numbers of inference steps and over all possible sequences for resolving probabilistic choices, using \wp we filter out the terminal states σ , and finally sum up the values of $\wp(\sigma, v)$.

For $\Pr_P(\downarrow)$ we basically do the same thing but we merely sum up the probabilities of reaching final states, thus ignoring the variable valuations, by using α instead of \wp . Regarding the termination probability of a probabilistic program, the case of almost-sure termination is of special interest:

Definition 7 (ALMOST-SURE TERMINATION):

We say that a program P **terminates almost-surely** iff $\Pr_P(\downarrow) = 1$. Consequently, we define the according set $\mathcal{AST} \subset \text{Prog}$ as

$$P \in \mathcal{AST} \iff \Pr_P(\downarrow) = 1.$$

In order to investigate the complexity of calculating $E_P(v)$, we define three sets: \mathcal{LEXP} , which relates to the set of lower bounds of $E_P(v)$, \mathcal{UEXP} , which relates to the set of upper bounds of $E_P(v)$, and \mathcal{EXP} which relates to $E_P(v)$ itself:

Definition 8 (\mathcal{LEXP} , \mathcal{UEXP} , AND \mathcal{EXP}):

The sets $\mathcal{LEXP}, \mathcal{UEXP}, \mathcal{EXP} \subseteq \text{Prog} \times \text{Var} \times \mathbb{Q}$ are defined as follows:

$$\begin{aligned} (P, v, q) \in \mathcal{LEXP} & \iff q < E_P(v) \\ (P, v, q) \in \mathcal{UEXP} & \iff q > E_P(v) \\ (P, v, q) \in \mathcal{EXP} & \iff q = E_P(v) \end{aligned}$$

5 Hardness of Computing Expected Outcomes

We now have all definitions available to begin the investigation of the computational hardness of computing expected outcomes. The first fact we observe is that lower bounds for expected outcomes are recursively enumerable:

Lemma 2:

$\mathcal{LEXP} \in \Sigma_1^0$, thus \mathcal{LEXP} is recursively enumerable.

Proof:

$$\begin{aligned} (P, v, q) \in \mathcal{LEXP} & \\ \iff q < E_P(v) & \\ \iff q < \sum_{i \in \mathbb{N}} \sum_{j \in \mathbb{N}} \wp \left(T_{prob}^* (\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v \right) & \\ \iff \exists y_1 \exists y_2: q < \sum_{i=0}^{y_1} \sum_{j=0}^{y_2} \wp \left(T_{prob}^* (\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v \right) & \\ \implies \mathcal{LEXP} \in \Sigma_1^0 & \quad \square \end{aligned}$$

Recursive enumerability of \mathcal{LEXP} means that lower bounds for expected outcomes can be effectively enumerated by some algorithm.

Now, if the set of upper bounds, i.e., \mathcal{UEXP} , was recursively enumerable as well, then expected outcomes would be computable reals. However, the contrary will be established over the course of the following lemmas:

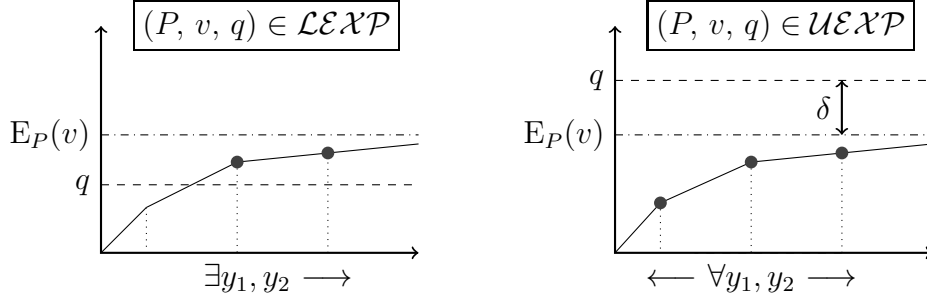


Figure 1: Schematic depiction of the formulae defining \mathcal{LEX} and \mathcal{UEX} , respectively. In each diagram, the solid line represents the monotonically increasing graph of $\sum_{0 \leq i \leq y_1} \sum_{0 \leq j \leq y_2} \wp(T_{prob}^*(\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v)$ plotted over increasing y_1 and y_2 .

Lemma 3:

$$\mathcal{UEX} \in \Sigma_2^0.$$

Proof:

$$\begin{aligned}
& (P, v, q) \in \mathcal{UEX} \\
& \iff q > E_P(v) \\
& \iff q > \sum_{i \in \mathbb{N}} \sum_{j \in \mathbb{N}} \wp(T_{prob}^*(\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v) \\
& \iff \exists \delta > 0 \forall y_1 \forall y_2: q - \delta > \sum_{i=0}^{y_1} \sum_{j=0}^{y_2} \wp(T_{prob}^*(\langle P, \eta_0, 1, \varepsilon \rangle, j, h(i)), v) \\
& \implies \mathcal{UEX} \in \Sigma_2^0
\end{aligned}$$

□

Figure 1 shows a schematic depiction of the intuition behind the formulae defining \mathcal{LEX} and \mathcal{UEX} , respectively.

After establishing $\mathcal{UEX} \in \Sigma_2^0$ there is in principle still hope that \mathcal{UEX} is recursively enumerable as $\Sigma_1^0 \subset \Sigma_2^0$. We will, however, establish next that $\mathcal{UEX} \in \Sigma_2^0 \setminus \Pi_2^0 \not\subseteq \Sigma_1^0$ meaning that \mathcal{UEX} is much harder to solve than, for instance, the halting problem. To establish this, we will make use of a well-known Π_2^0 -complete problem, namely the *universal* halting problem for ordinary programs.

Definition 9 (THE UNIVERSAL HALTING PROBLEM):

The **universal halting problem** is a subset $\mathcal{UH} \subset \text{ordProg}$, which is characterized as follows:

$$P \in \mathcal{UH} : \iff \forall \eta \exists k \exists \eta': \langle P, \eta, 1, \varepsilon \rangle \vdash^k \langle \downarrow, \eta', 1, \varepsilon \rangle$$

We denote by $\overline{\mathcal{UH}}$ the **complement** of \mathcal{UH} , i.e., $\overline{\mathcal{UH}} = \text{ordProg} \setminus \mathcal{UH}$.

In other words, a program P is in \mathcal{UH} , if it terminates its computation after a finite number of steps starting in *any* initial valuation η . A characterization from a more computational point of view would be that $P \in \mathcal{UH}$ if and only if P satisfies $\forall \eta \exists k \exists \eta': T_{prob}^*(\langle P, \eta, 1, \varepsilon \rangle, k, \varepsilon) = \langle \downarrow, \eta', 1, \varepsilon \rangle$.

The universal halting problem and its complement satisfy the following completeness properties:

Theorem 2 ([15]):

\mathcal{UH} is Π_2^0 -complete and $\overline{\mathcal{UH}}$ is Σ_2^0 -complete.

Next we will exploit Theorem 2 to establish the Σ_2^0 -completeness of \mathcal{UEXP} :

Lemma 4:

\mathcal{UEXP} is Σ_2^0 -complete.

Proof: By Lemma 3 we have $\mathcal{UEXP} \in \Sigma_2^0$, so it remains to show that \mathcal{UEXP} is Σ_2^0 -hard: We do this by proving $\overline{\mathcal{UH}} \leq_m \mathcal{UEXP}$. Consider the following function $f: \overline{\mathcal{UH}} \leq_m \mathcal{UEXP}$: f takes an ordinary program $Q \in \text{ordProg}$ as its input and returns the triple $(P, v, 1)$, where v does not occur in Q and $P \in \text{Prog}$ is the following probabilistic program:

```

i := 0; {continue := 0} [0.5] {continue := 1};
while (continue ≠ 0){
  i := i + 1;
  {continue := 0} [0.5] {continue := 1}
};
s := 0; {continue := 0} [0.5] {continue := 1};
while (continue ≠ 0){
  s := s + 1;
  {continue := 0} [0.5] {continue := 1}
};
v := 0; TQ

```

TQ is a program that computes $\wp(\text{T}_{\text{prob}}^*(\langle Q; v := 1, g_Q(i), 1, \varepsilon \rangle, s, \varepsilon), v) \cdot 2^{s+1}$ and stores the result in the variable v , and $g_Q: \mathbb{N} \rightarrow \mathbb{V}$ is some computable bijection, such that $\forall z \in \text{Var}: [g_Q(i)](z) \neq 0$ implies that z occurs in Q .

Partial Correctness: $\wp(\text{T}_{\text{prob}}^*(\langle Q; v := 1, g_Q(i), 1, \varepsilon \rangle, s, \varepsilon), v) \cdot 2^{s+1}$ returns 2^{s+1} if and only if Q halts on input $g_Q(i)$ after exactly s steps (otherwise 0), because only then, the variable v is set to 1 after executing the program $Q; v := 1$ for s steps. The two while-loops generate independent geometric distributions with parameter 0.5 on i and s , respectively, so the probability of generating exactly the numbers i and s is $(2^i \cdot 2^s)^{-1}$. The expected value of v after executing the program P is hence

$$\sum_{i \in \mathbb{N}} \sum_{s \in \mathbb{N}} \frac{1}{2^i \cdot 2^s} \cdot \wp\left(\text{T}_{\text{prob}}^*(\langle Q; v := 1, g_Q(i), 1, \varepsilon \rangle, s+1, \varepsilon), v\right) \cdot 2^{s+1}.$$

Since for each input, the number of steps until termination is either unique or does not exist, the formula for the expected outcome reduces to $\sum_{i \in \mathbb{N}} 2^{-i} \cdot 2 = 1$ if and only if Q halts on every input after some finite number of steps. Thus if there exists an input on which Q *does not* eventually halt, then $(P, v, 1) \in \mathcal{UEXP}$ as then the expected value is strictly less than one. If, on the other hand, Q *does* halt on every input, then the expected value is exactly one and hence $(P, v, 1) \notin \mathcal{UEXP}$.

Total Correctness: It is an easy but tedious exercise to construct a program computing $g_Q(i)$ given only Q . Program code for \wp , T_{prob}^* , multiplication and potentiation is also computable. So in total, the program code for P and thereby the triple $(P, v, 1)$ is computable.

By Theorem 2, $\overline{\mathcal{UH}}$ is Σ_2^0 -complete, so for any $\mathcal{A} \in \Sigma_2^0$ it holds that $\mathcal{A} \leq_m \overline{\mathcal{UH}}$. Since we have just proven that $\overline{\mathcal{UH}} \leq_m \mathcal{UEXP}$, it follows that $\mathcal{A} \leq_m \overline{\mathcal{UH}} \leq_m \mathcal{UEXP}$, and by transitivity $\mathcal{A} \leq_m \mathcal{UEXP}$. \square

Finally, it follows from Lemma 1 that membership for \mathcal{UEXP} is in some sense the hardest problem in Σ_2^0 :

Corollary 4:

$$\mathcal{UEXP} \in \Sigma_2^0 \setminus \Pi_2^0.$$

Remark 1:

In the probabilistic program P used in the proof of Lemma 4, the randomization and the actual computation are completely separated. Since P in a sense “solves” the universal halting problem, we interpret this as possible evidence that P is in some sort of “normal form” for probabilistic programs.

We now go on with characterizing the complexity of the set \mathcal{EXP} , which is the set we are mainly interested in, when it comes to expected outcomes. As a first result we establish the following:

Lemma 5:

$$\mathcal{EXP} \in \Pi_2^0.$$

Proof: By Lemma 3, there exists a decidable relation \mathcal{U} , such that $(P, v, x) \in \mathcal{UEXP}$ iff $\exists r_1 \forall r_2: (r_1, r_2, P, v, x) \in \mathcal{U}$. Furthermore from Lemma 2 it follows that there exists a decidable relation \mathcal{L} , such that $(P, v, x) \in \mathcal{LEXP} \iff \exists \ell: (\ell, P, v, x) \in \mathcal{L}$. Let $\neg\mathcal{U}$ and $\neg\mathcal{L}$ be the (decidable) negations of \mathcal{U} and \mathcal{L} , respectively, then:

$$\begin{aligned} & (P, v, q) \in \mathcal{EXP} \\ \iff & q = E_P(v) \\ \iff & q \leq E_P(v) \wedge q \geq E_P(v) \\ \iff & \neg(q > E_P(v)) \wedge \neg(q < E_P(v)) \\ \iff & \neg(\exists r_1 \forall r_2: (r_1, r_2, P, v, q) \in \mathcal{U}) \wedge \neg(\exists \ell: (\ell, P, v, q) \in \mathcal{L}) \\ \iff & (\forall r_1 \exists r_2: (r_1, r_2, P, v, q) \in \neg\mathcal{U}) \wedge (\forall \ell: (\ell, P, v, q) \in \neg\mathcal{L}) \\ \iff & \forall r_1 \forall \ell \exists r_2: (r_1, r_2, P, v, q) \in \neg\mathcal{U} \wedge (\ell, P, v, q) \in \neg\mathcal{L} \\ \implies & \mathcal{EXP} \in \Pi_2^0 \quad \square \end{aligned}$$

Intuitively the above proof asserts that we check whether $q = E_P(v)$ by deciding both $q \leq E_P(v)$ and $q \geq E_P(v)$ and that this check can be done by deciding a Π_2^0 -relation. Furthermore, we now establish the main theorem showing that \mathcal{EXP} is Π_2^0 -complete, thus extremely hard to solve:

Theorem 3:

\mathcal{EXP} is Π_2^0 -complete.

Proof: By Lemma 5, $\mathcal{EXP} \in \Pi_2^0$, so it remains to show that \mathcal{EXP} is Π_2^0 -hard. We do this by proving $\mathcal{UH} \leq_m \mathcal{EXP}$. Consider again the function f from the proof of Lemma 4: Given an ordinary program Q , f computes the triple $(P, v, 1)$, where P is a probabilistic program P which has an expected outcome of one for the variable v if and only if Q terminates on all inputs, which is nothing else than $Q \in \mathcal{UH}$. Thus $f: \mathcal{UH} \leq_m \mathcal{EXP}$.

By Theorem 2, \mathcal{UH} is Π_2^0 -complete, so for any $\mathcal{A} \in \Pi_2^0$ it holds that $\mathcal{A} \leq_m \mathcal{UH}$. Since we have just proven that $\mathcal{UH} \leq_m \mathcal{EXP}$, it follows that $\mathcal{A} \leq_m \mathcal{UH} \leq_m \mathcal{EXP}$, and by transitivity $\mathcal{A} \leq_m \mathcal{EXP}$. \square

It now follows from Lemma 1 that membership for \mathcal{EXP} is in some sense the hardest problem in Π_2^0 :

Corollary 5:

$$\mathcal{E}\mathcal{X}\mathcal{P} \in \Pi_2^0 \setminus \Sigma_2^0 .$$

6 Hardness of Deciding Almost–Sure Termination

In this section, we turn towards the problem of almost–sure termination and establish completeness results for this problem. We first establish that almost–sure termination is many–one reducible to $\mathcal{E}\mathcal{X}\mathcal{P}$ and thereby lays in Π_2^0 :

Lemma 6:

$$\mathcal{A}\mathcal{S}\mathcal{T} \leq_m \mathcal{E}\mathcal{X}\mathcal{P} .$$

Proof: Consider the following function f which takes a probabilistic program Q as its input and returns the triple $(P, v, 1)$, where P is the following probabilistic program:

$v := 0; Q; v := 1$

Total and Partial Correctness: The triple $(P, v, 1)$ is obviously computable. On executing P , the variable v is set to one only in those runs in which the program Q terminates. So the expected value of v converges to one, if and only if the probability of Q terminating converges to one. So if $Q \in \mathcal{A}\mathcal{S}\mathcal{T}$, then and only then $(P, v, 1) \in \mathcal{E}\mathcal{X}\mathcal{P}$. Thus $f: \mathcal{A}\mathcal{S}\mathcal{T} \leq_m \mathcal{E}\mathcal{X}\mathcal{P}$. \square

By Theorem 3, $\mathcal{E}\mathcal{X}\mathcal{P}$ is Π_2^0 –complete, so it follows directly from Lemma 6 that:

Corollary 6:

$$\mathcal{A}\mathcal{S}\mathcal{T} \in \Pi_2^0 .$$

Next we will establish the Π_2^0 –hardness and thereby the Π_2^0 –completeness of $\mathcal{A}\mathcal{S}\mathcal{T}$ by many–one–reduction from the universal halting problem:

Theorem 4:

$\mathcal{A}\mathcal{S}\mathcal{T}$ is Π_2^0 –complete.

Proof: By Corollary 6, $\mathcal{A}\mathcal{S}\mathcal{T} \in \Pi_2^0$, so it remains to show that $\mathcal{A}\mathcal{S}\mathcal{T}$ is Π_2^0 –hard. For that we many–one reduce the Π_2^0 –complete universal halting problem to $\mathcal{A}\mathcal{S}\mathcal{T}$ using the following function $f: \mathcal{U}\mathcal{H} \leq_m \mathcal{A}\mathcal{S}\mathcal{T}$: f takes an ordinary program Q as its input and returns the following probabilistic program P :

```

i := 0; {continue := 0} [0.5] {continue := 1};
while (continue ≠ 0){
  i := i + 1;
  {continue := 0} [0.5] {continue := 1}
};
TQ

```

where TQ is an ordinary program that simulates the program Q on input $g_Q(i)$, and $g_Q: \mathbb{N} \rightarrow \mathbb{V}$ is some computable bijection, such that $\forall v \in \mathbf{Var}: [g_Q(i)](v) \neq 0$ implies that v occurs in Q .

Partial Correctness: The while–loop in P establishes a geometric distribution with parameter 0.5 on i and hence a geometric distribution on all possible inputs for Q . After the while–loop, the program Q is simulated on the input generated probabilistically in the

while-loop. Obviously then the entire program P terminates with probability one, i.e., terminates almost-surely, if and only if the simulation of Q terminates on every input. Thus $Q \in \mathcal{UH}$ if and only if $P \in \mathcal{AST}$.

Total Correctness: As mentioned in the proof of Lemma 4, the program code for g_Q is computable. Also the program code for a universal program capable of simulating any program Q on a given input is computable [9]. So in total, the program code for P is computable.

By Theorem 2, \mathcal{UH} is Π_2^0 -complete, so for any $\mathcal{A} \in \Pi_2^0$ it holds that $\mathcal{A} \leq_m \mathcal{UH}$. Since we have just proven that $\mathcal{UH} \leq_m \mathcal{AST}$, it follows that $\mathcal{A} \leq_m \mathcal{UH} \leq_m \mathcal{AST}$, and by transitivity $\mathcal{A} \leq_m \mathcal{AST}$. \square

7 Conclusion

Our results show that one can effectively enumerate all rationals that are strictly less than the expected outcome for a program variable v after executing a probabilistic program P , i.e., arbitrarily close approximations from below are computable. Obtaining such approximations from above is harder: These would be recursively enumerable only if there would be access to an oracle for the (non-universal) halting problem [9, 14]. This approximation problem is as hard to solve as deciding whether e.g., an ordinary program halts on finitely many inputs [15].

Deciding almost-sure termination is even harder and is as hard as computing exact expected outcomes. Namely, such outcomes are not recursively enumerable even if there would be access to an oracle for the halting problem [15]. Other natural examples that are equally hard are the universal halting problem and the problem of deciding whether an ordinary program halts on infinitely many inputs [15].

The established hardness results give insights into the specific difficulties of dealing with the studied decision problems. In particular further research could be directed towards identifying subsets of probabilistic programs for which the upper bounds of the expected outcome are given by a Σ_2^0 -set $\mathcal{A} = \{x \mid \exists y_1 \forall y_2: (x, y_1, y_2) \in R\}$ such that the set $\mathcal{A}' = \{(x, y_1) \mid \forall y_2: (x, y_1, y_2) \in R\}$ is decidable. In this case, the set \mathcal{A} of upper bounds would be recursively enumerable and thus the *exact* expected outcome can be approximated arbitrarily close from below *and* from above. Obtaining and deciding \mathcal{A}' would basically amount to transforming a given probabilistic program into an ordinary program for which then a non-termination proof has to be found which in certain cases can be automated.

Aside from the above considerations the structure of the probabilistic programs we use in our proofs hints towards the possible existence of a normal form for probabilistic programs in which the randomization and the actual computation are separated. Further investigation of this issue is planned. Further research could also deal with hardness of the considered problems in presence of non-determinism.

References

- [1] Tamarah Arons, Amir Pnueli, and Lenore D. Zuck. Parameterized Verification by Probabilistic Abstraction. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of

- LNCS*, pages 87–102. Springer, 2003.
- [2] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9, 2013.
 - [3] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure Transformer Semantics for Bayesian Machine Learning. *Logical Methods in Computer Science*, 9(3), 2013.
 - [4] Olivier Bournez and Mathieu Hoyrup. Rewriting Logic and Probabilities. In Robert Nieuwenhuis, editor, *RTA*, volume 2706 of *LNCS*, pages 61–75. Springer, 2003.
 - [5] Martin David Davis. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 1994.
 - [6] Javier Esparza, Andreas Gaiser, and Stefan Kiefer. Proving Termination of Probabilistic Programs Using Patterns. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 123–138. Springer, 2012.
 - [7] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus Weakest Pre-Expectation Semantics for the Probabilistic Guarded Command Language. *Performance Evaluation*, 73:110–132, 2014.
 - [8] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of Probabilistic Concurrent Programs. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, 1983.
 - [9] Stephen Cole Kleene. Recursive Predicates and Quantifiers. *Trans. of the AMS*, 53(1):41 – 73, 1943.
 - [10] Dexter Kozen. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
 - [11] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2004.
 - [12] Carroll Morgan. Proof Rules for Probabilistic Loops. In *Proceedings of the BCS-FACS 7th Conference on Refinement*, FAC-RW’96, page 10, Swinton, UK, 1996. British Computer Society.
 - [13] Andrzej S. Murawski and Joël Ouaknine. On Probabilistic Program Equivalence and Refinement. In *CONCUR 2005 – Concurrency Theory*, volume 3653 of *LNCS*, pages 156–170. Springer, 2005.
 - [14] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier, 1992.
 - [15] Piergiorgio Odifreddi. *Classical Recursion Theory, Volume II*. Elsevier, 1999.
 - [16] Emil Leon Post. Recursively Enumerable Sets of Positive Integers and their Decision Problems. *Bulletin of the AMS*, 50(5):284–316, 1944.
 - [17] Jon Sneyers and Danny De Schreye. Probabilistic Termination of CHRiSM Programs. In Germán Vidal, editor, *LOPSTR*, volume 7225 of *LNCS*, pages 221–236. Springer, 2011.